Disclaimer & Acknowledgement

Slides have been prepared using/adapting/modifying a number of resources, including pptx and images, available online. We deeply acknowledge faculty/researchers posting these resources online.

Convolutional Neural Networks

- Class of deep neural networks
- Highly effective for tasks such as image classification, object detection, and semantic segmentation.

- Class of deep neural networks
- Highly effective for tasks such as image classification, object detection, and semantic segmentation.

Q. Classification vs Localization vs Object Detection vs Semantic Segmentation vs Instance Segmentation.

- Class of deep neural networks
- Highly effective for tasks such as image classification, object detection, and semantic segmentation.

Q. Classification vs Localization vs Object Detection vs Semantic Segmentation vs Instance Segmentation vs Panoptic Segmentation

Classification -> Classify entire image to one class

Localization -> Locate a single object within the image (bbox)

Object Detection -> Identify and locate multiple objects within an image

Semantic Segmentation -> Classify each pixel to a particular class or background

Instance Segmentation -> Only concerned with the objects (not background), classifies into categories on the basis of “instances”.

Panoptic Segmentation -> Assign a class label to each pixel while differentiating between the different instances of the same class

Recap of Important Concepts and Components of CNNs

Recap of Important Concepts and Components of CNNs

Convolution Operation

Recap of Important Concepts and Components of CNNs

Convolution Operation

Used to extract features from the input image by applying a filter (kernel) across the spatial dimensions.

Recap of Important Concepts and Components of CNNs

Convolution Operation

Used to extract features from the input image by applying a filter (kernel) across the spatial dimensions.

A filter (e.g., 3x3 or 5x5 matrix) slides over the input image, performing element-wise multiplication and summing the results to produce a feature map.
This process is repeated for multiple filters to capture different features.

Recap of Important Concepts and Components of CNNs

Padding

Recap of Important Concepts and Components of CNNs

Padding

Used to control the spatial dimensions of the output feature maps.

Recap of Important Concepts and Components of CNNs

Padding

Used to control the spatial dimensions of the output feature maps.

Common padding types include 'valid' (no padding) and 'same' (padding to ensure the output has the same dimensions as the input).
Helps avoid reducing the size of the feature maps after each convolution operation.

Recap of Important Concepts and Components of CNNs

Stride

Recap of Important Concepts and Components of CNNs

Stride

The step size with which the filter moves across the input image.
A stride of 1 moves the filter one pixel at a time, while a stride of 2 moves two pixels at a time.

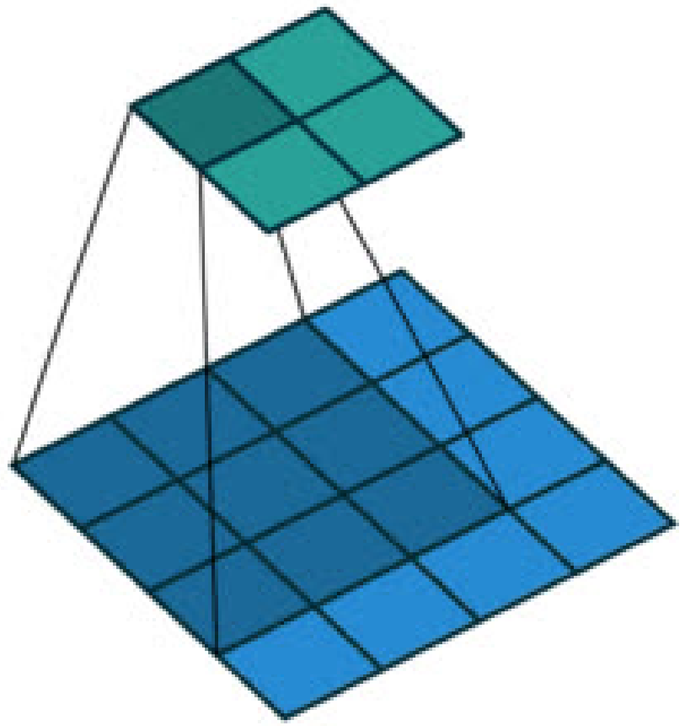
Recap of Important Concepts and Components of CNNs

Stride

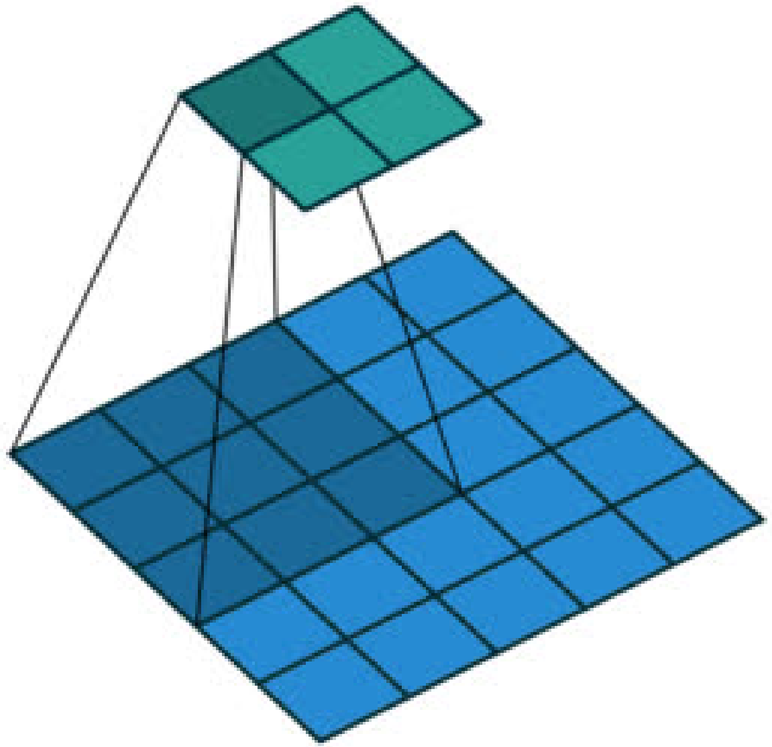
The step size with which the filter moves across the input image.
A stride of 1 moves the filter one pixel at a time, while a stride of 2 moves two pixels at a time.

Reduces the spatial dimensions of the output feature map.
Reduces the computational load.

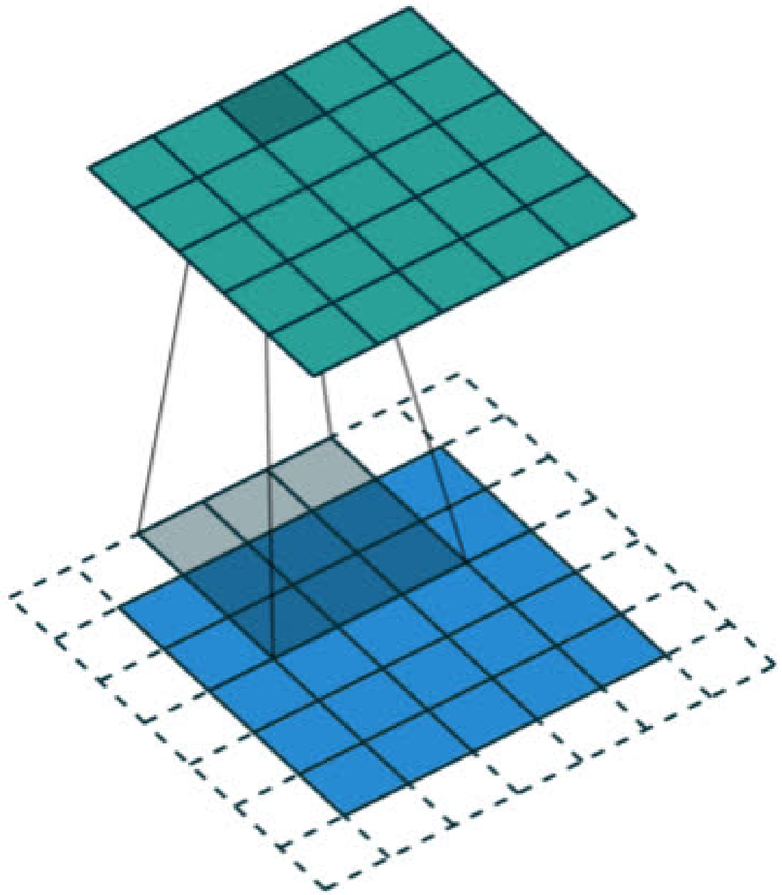
Convolution (No Padding, Stride = 1)



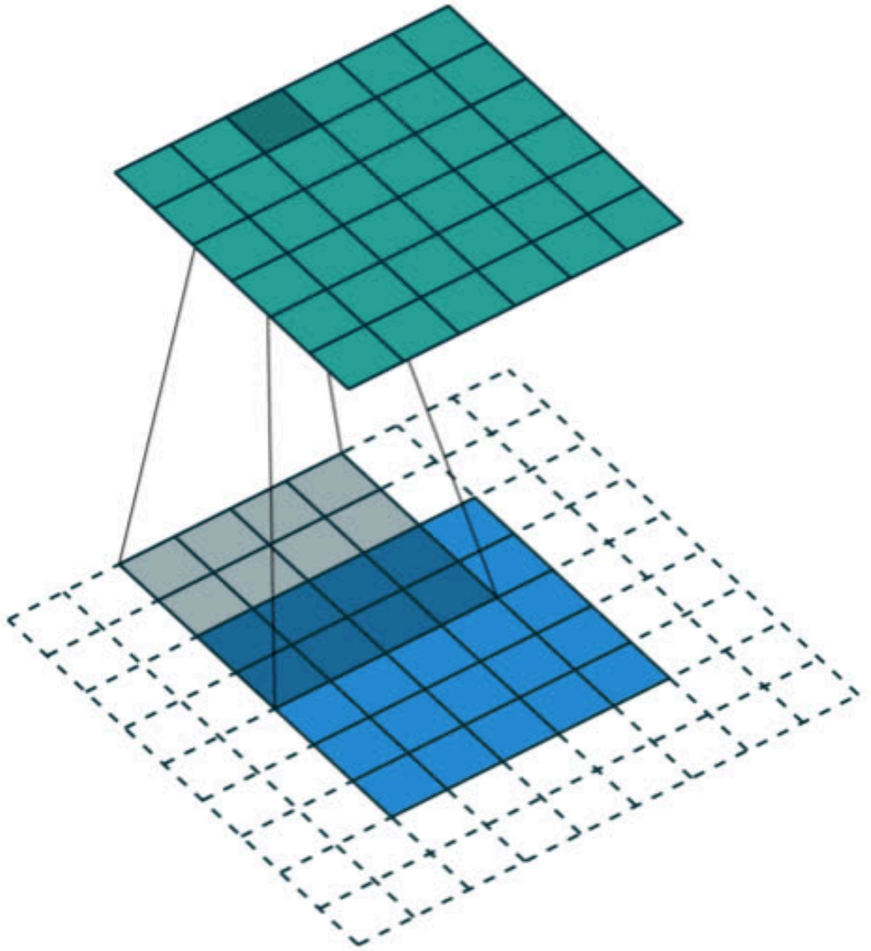
Convolution (No Padding, Stride = 2)



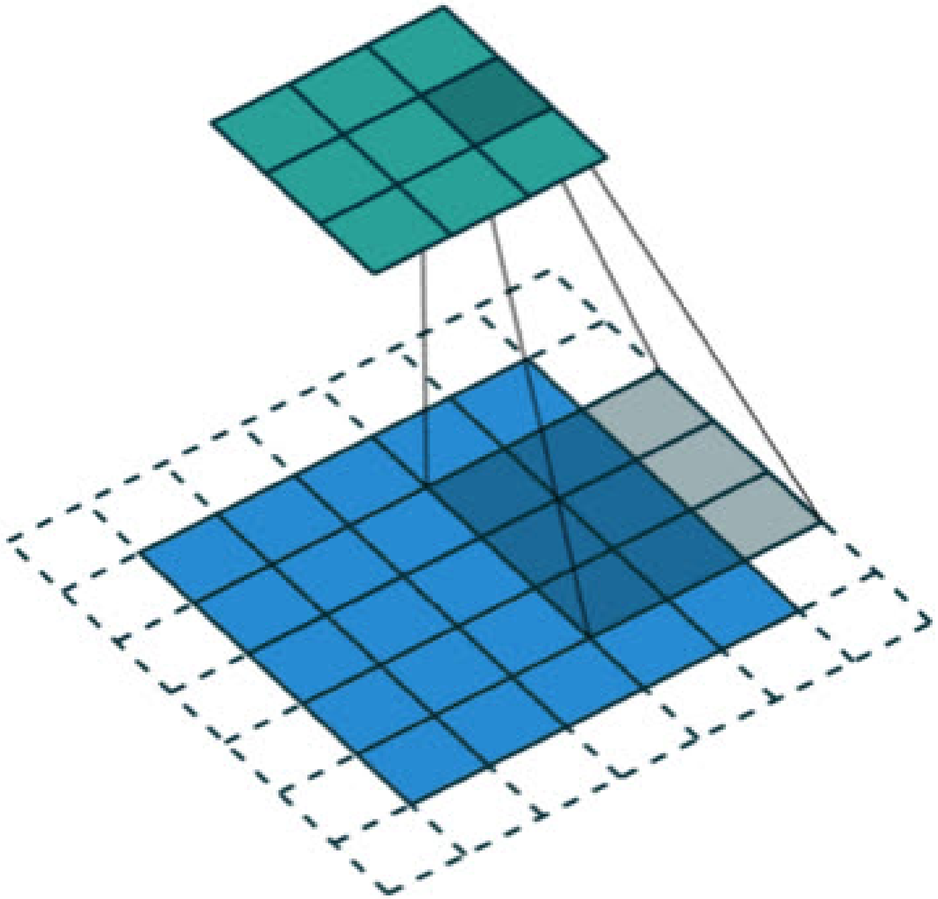
Convolution (Padding = 1, Stride = 1)



Convolution (Padding = 2, Stride = 1)



Convolution (Padding = 1, Stride = 2)



Recap of Important Concepts and Components of CNNs

Transposed Convolution

Recap of Important Concepts and Components of CNNs

Transposed Convolution

Used to increase the spatial dimensions of the input feature map, effectively performing upsampling.
Often used in decoder part of architectures.

Recap of Important Concepts and Components of CNNs

Transposed Convolution

Used to increase the spatial dimensions of the input feature map, effectively performing upsampling.
Often used in decoder part of architectures.

Also referred to as deconvolutions (but the term is not preferred)

Recap of Important Concepts and Components of CNNs

Transposed Convolution

Recap of Important Concepts and Components of CNNs

Dilated Convolution

Recap of Important Concepts and Components of CNNs

Dilated Convolution

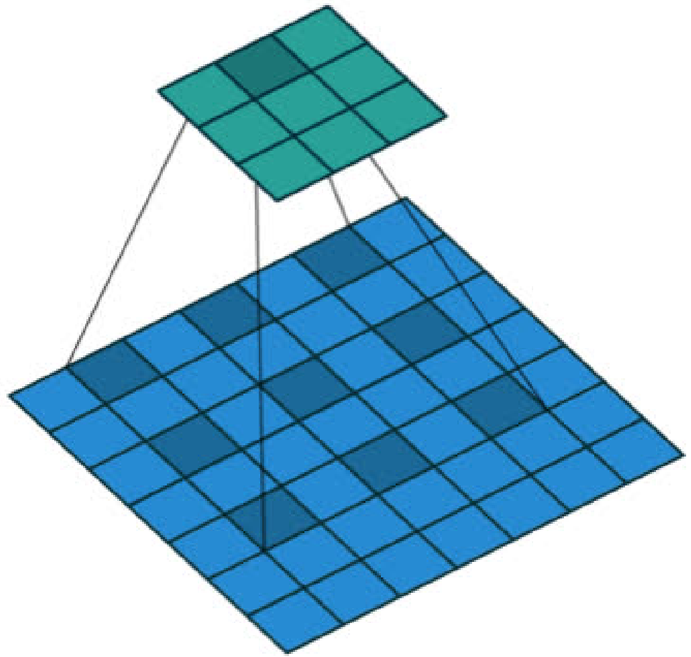
introduces gaps (dilations) between the elements of the kernel.

This technique allows the convolutional layer to have a larger receptive field without increasing the number of parameters or the amount of computation significantly.

Also known as atrous convolutions.

Recap of Important Concepts and Components of CNNs

Dilated Convolution



Recap of Important Concepts and Components of CNNs

Convolutional Layer (conv2d)

Applies a 2D convolution operation over an input signal composed of several input planes.

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int, tuple or str, optional*) – Padding added to all four sides of the input. Default: 0
- **padding_mode** (*str, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

```python
import torch
import torch.nn as nn

conv_layer = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)

input_tensor = torch.randn(1, 3, 32, 32)

output_tensor = conv_layer(input_tensor)

print("Input shape:", input_tensor.shape)
print("Output shape:", output_tensor.shape)
```

```
Input shape: torch.Size([1, 3, 32, 32])
Output shape: torch.Size([1, 16, 32, 32])
```

Input & Output Dimensions

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Q. Input Dimensions - 227 x 227 x 3

First Layer - 96 filters (11 x 11) applied at stride 4

What is the size of the output volume? Or what are the output dimensions?

Q. Input Dimensions - 227 x 227 x 3

First Layer - 96 filters (11 x 11) applied at stride 4

What is the size of the output volume? Or what are the output dimensions?

Ans = 55 x 55 x 96

Q. Input Dimensions - 227 x 227 x 3

First Layer - 96 filters (11 x 11) applied at stride 4

What is the size of the output volume? Or what are the output dimensions?

Ans = 55 x 55 x 96

Q. What is the total no. of parameters in this layer?

Q. Input Dimensions - 227 x 227 x 3

First Layer - 96 filters (11 x 11) applied at stride 4

What is the size of the output volume? Or what are the output dimensions?

Ans = 55 x 55 x 96

Q. What is the total no. of parameters in this layer?

Ans = (11 x 11 x 3 + 1) x 96

Q. Input Dimensions - 227 x 227 x 3

After Conv1: 55 x 55 x 96

Second Layer : Pool1 : 3 x 3 filters applied at stride 2

What is the output volume size? Or what are the output dimensions?

Q. Input Dimensions - 227 x 227 x 3

After Conv1: 55 x 55 x 96

Second Layer : Pool1 : 3 x 3 filters applied at stride 2

What is the output volume size? Or what are the output dimensions?

Ans = 27 x 27 x 96

Q. Input Dimensions - 227 x 227 x 3

After Conv1: 55 x 55 x 96

Second Layer : Pool1 : 3 x 3 filters applied at stride 2

What is the output volume size? Or what are the output dimensions?

Ans = 27 x 27 x 96

Q. What is the total no. of parameters in this layer?

Q. Input Dimensions - 227 x 227 x 3

After Conv1: 55 x 55 x 96

Second Layer : Pool1 : 3 x 3 filters applied at stride 2

What is the output volume size? Or what are the output dimensions?

Ans = 27 x 27 x 96

Q. What is the total no. of parameters in this layer?

Ans = 0

Recap of Important Concepts and Components of CNNs

Transposed Convolutional Layer (convTranspose2d)

Applies a 2D transposed convolution operator over an input signal composed of several input planes.

CLASS torch.nn.ConvTranspose2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Default: 0
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1

```python
import torch
import torch.nn as nn

conv_transpose = nn.ConvTranspose2d(in_channels=3, out_channels=16, kernel_size=3, stride=2, padding=1)

input_tensor = torch.randn(1, 3, 64, 64)

output_tensor = conv_transpose(input_tensor)

print(output_tensor.shape)
```

```
torch.Size([1, 16, 127, 127])
```

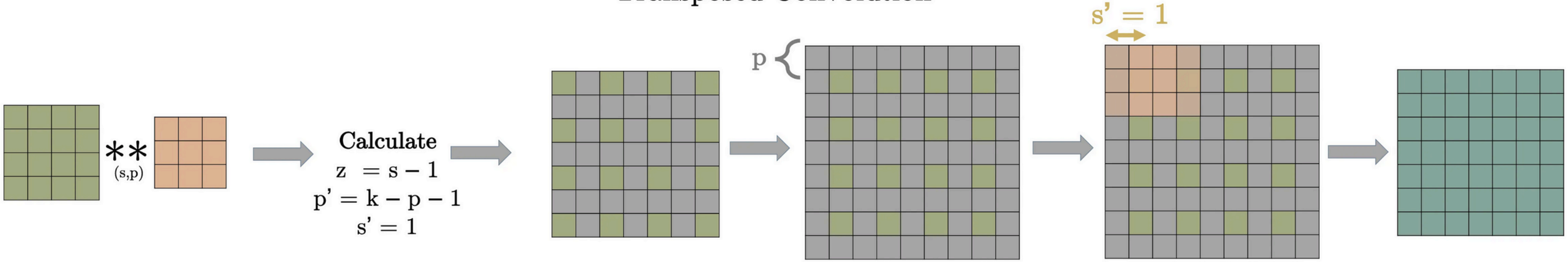
Input & Output Dimensions

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + 1$$

Transposed Convolution



Recap of Important Concepts and Components of CNNs

Max Pooling Layer (MaxPool2D)

Applies a 2D max pooling over an input signal composed of several input planes.

CLASS torch.nn.MaxPool2d(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*) [SOURCE]

- **kernel_size** (*Union*[*int*, *Tuple*[*int*, *int*]]) – the size of the window to take a max over
- **stride** (*Union*[*int*, *Tuple*[*int*, *int*]]) – the stride of the window. Default value is `kernel_size`
- **padding** (*Union*[*int*, *Tuple*[*int*, *int*]]) – Implicit negative infinity padding to be added on both sides
- **dilation** (*Union*[*int*, *Tuple*[*int*, *int*]]) – a parameter that controls the stride of elements in the window
- **return_indices** (*bool*) – if `True`, will return the max indices along with the outputs. Useful for `torch.nn.MaxUnpool2d` later
- **ceil_mode** (*bool*) – when True, will use *ceil* instead of *floor* to compute the output shape

```python
import torch
import torch.nn as nn

conv_transpose = nn.MaxPool2d(kernel_size=2, stride=2)

input_tensor = torch.randn(1, 3, 5, 5)

output_tensor = conv_transpose(input_tensor)

print(output_tensor.shape)
```

```
torch.Size([1, 3, 2, 2])
```

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Recap of Important Concepts and Components of CNNs

Max UnPooling Layer (MaxUnPool2d)

Computes a partial inverse of MaxPool2d.

MaxPool2d is not fully invertible since the non-maximal values are lost.

CLASS torch.nn.MaxUnpool2d(*kernel_size*, *stride=None*, *padding=0*) [SOURCE]

- **kernel_size** (*int or tuple*) – Size of the max pooling window.
- **stride** (*int or tuple*) – Stride of the max pooling window. It is set to `kernel_size` by default.
- **padding** (*int or tuple*) – Padding that was added to the input

```python
import torch
import torch.nn as nn

pool = nn.MaxPool2d(2, stride=2, return_indices=True)
unpool = nn.MaxUnpool2d(2, stride=2)
input = torch.tensor([[[[ 1.,  2.,  3.,  4.],
                        [ 5.,  6.,  7.,  8.],
                        [ 9., 10., 11., 12.],
                        [13., 14., 15., 16.]]]])

output, indices = pool(input)
unpool(output, indices)
```

```
tensor([[[[ 0.,  0.,  0.,  0.],
          [ 0.,  6.,  0.,  8.],
          [ 0.,  0.,  0.,  0.],
          [ 0., 14.,  0., 16.]]]])
```

Recap of Important Concepts and Components of CNNs

Average Pooling Layer(AvgPool2d)

Applies a 2D average pooling over an input signal composed of several input planes.

CLASS torch.nn.AvgPool2d(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*) [SOURCE]

- **kernel_size** (*Union*[*int*, *Tuple*[*int*, *int*]]) – the size of the window
- **stride** (*Union*[*int*, *Tuple*[*int*, *int*]]) – the stride of the window. Default value is `kernel_size`
- **padding** (*Union*[*int*, *Tuple*[*int*, *int*]]) – implicit zero padding to be added on both sides
- **ceil_mode** (*bool*) – when True, will use *ceil* instead of *floor* to compute the output shape
- **count_include_pad** (*bool*) – when True, will include the zero-padding in the averaging calculation
- **divisor_override** (*Optional*[*int*]) – if specified, it will be used as divisor, otherwise size of the pooling region will be used.

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

```python
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[1., 2., 3., 4., 5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])

avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)

output_tensor = avg_pool(input_tensor)

print("Input Tensor:\n", input_tensor)
print("Output Tensor after AvgPool2d:\n", output_tensor)
```

```
Input Tensor:
 tensor([[[[ 1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10.],
          [11., 12., 13., 14., 15.],
          [16., 17., 18., 19., 20.]]])
Output Tensor after AvgPool2d:
 tensor([[[[ 4.,  6.],
          [14., 16.]]]])
```

Recap of Important Concepts and Components of CNNs

Adaptive Average Pooling Layer(AdaptiveAvgPool2d)

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

CLASS torch.nn.AdaptiveAvgPool2d(*output_size*)

Parameters

output_size (*Union*[*int*, *None*, *Tuple*[*Optional*[*int*], *Optional*[*int*]]]) – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single H for a square image H x H. H and W can be either a `int`, or `None` which means the size will be the same as that of the input.

Shape:

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- Output: (N, C, S_0, S_1) or (C, S_0, S_1), where $S = \text{output\_size}$.

Adaptive pooling divides the input tensor into regions such that the output tensor has the specified dimensions.

You directly specify the desired output height and width.

For an input tensor with dimensions Hin×Win and an output tensor with dimensions Hout×Wout:

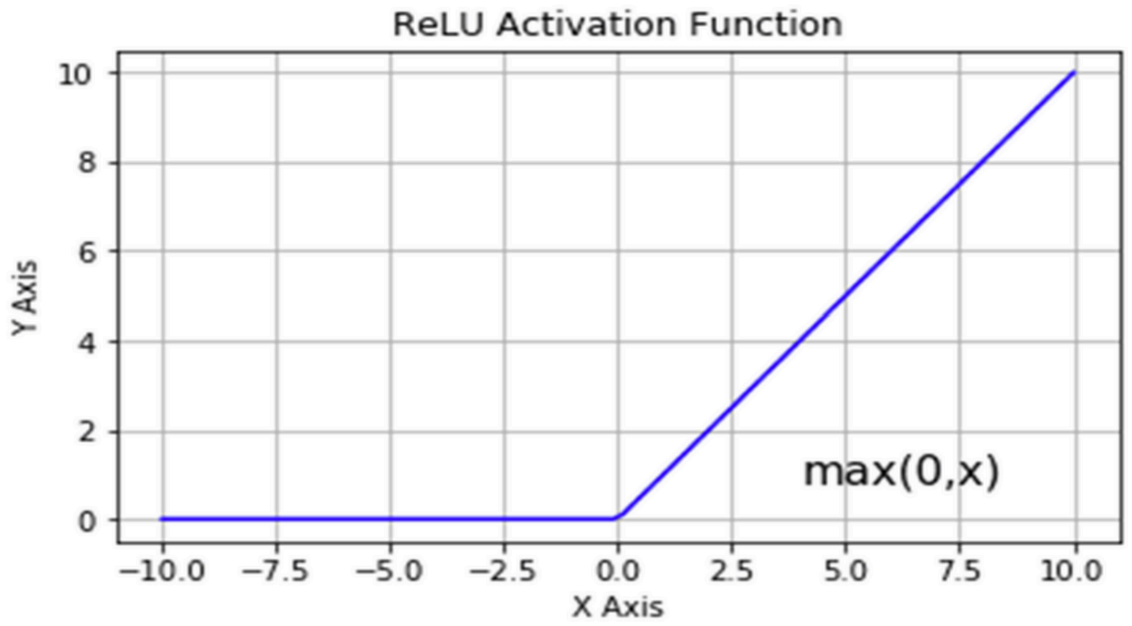
- The height of each pooling region is calculated as floor(Hin/Hout).
- The width of each pooling region is calculated as floor(Win/Wout).

```python
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[1., 2., 3., 4., 5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])

adaptive_pool = nn.AdaptiveAvgPool2d((2, 2))

output_tensor = adaptive_pool(input_tensor)
print("Input Tensor:\n", input_tensor)
print("Output Tensor:\n", output_tensor)
```

```
Input Tensor:
 tensor([[[[ 1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10.],
          [11., 12., 13., 14., 15.],
          [16., 17., 18., 19., 20.]]]])
Output Tensor:
 tensor([[[[ 4.5000,  6.5000],
          [14.5000, 16.5000]]]])
```

Recap of Important Concepts and Components of CNNs

ReLU (Rectified Linear Unit)

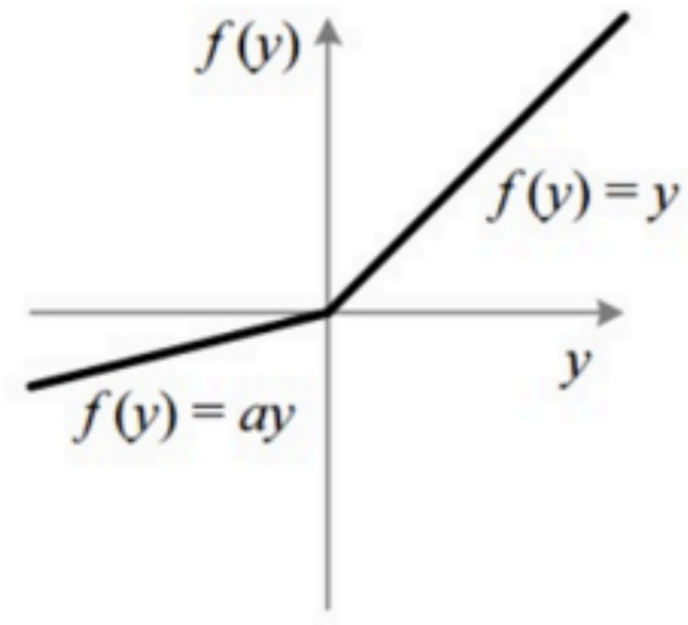



```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[-1., -2., -3., -4., -5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])


relu = nn.ReLU()
output = relu(input_tensor)
output
```

```
tensor([[[[ 0.,  0.,  0.,  0.,  0.],
          [ 6.,  7.,  8.,  9., 10.],
          [11., 12., 13., 14., 15.],
          [16., 17., 18., 19., 20.]]]])
```

Recap of Important Concepts and Components of CNNs

Leaky ReLU

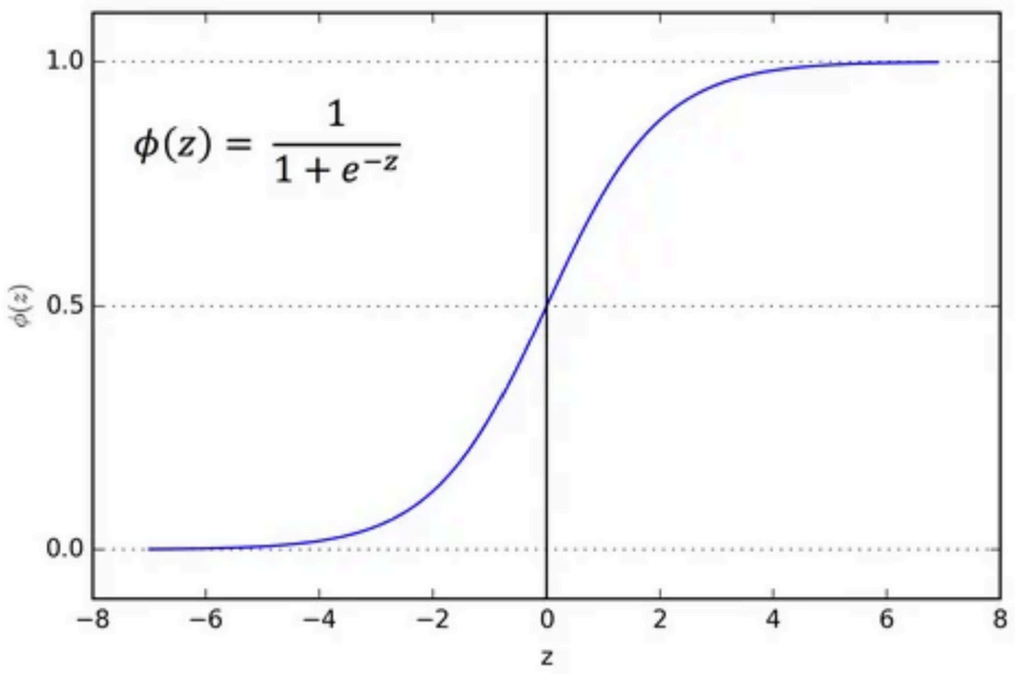



```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[-1., -2., -3., -4., -5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])

leaky_relu = nn.LeakyReLU(negative_slope=0.01)
output = leaky_relu(input_tensor)
output
```

```
tensor([[[[-1.0000e-02, -2.0000e-02, -3.0000e-02, -4.0000e-02, -5.0000e-02],
          [ 6.0000e+00,  7.0000e+00,  8.0000e+00,  9.0000e+00,  1.0000e+01],
          [ 1.1000e+01,  1.2000e+01,  1.3000e+01,  1.4000e+01,  1.5000e+01],
          [ 1.6000e+01,  1.7000e+01,  1.8000e+01,  1.9000e+01,  2.0000e+01]]]])
```

Recap of Important Concepts and Components of CNNs

Sigmoid

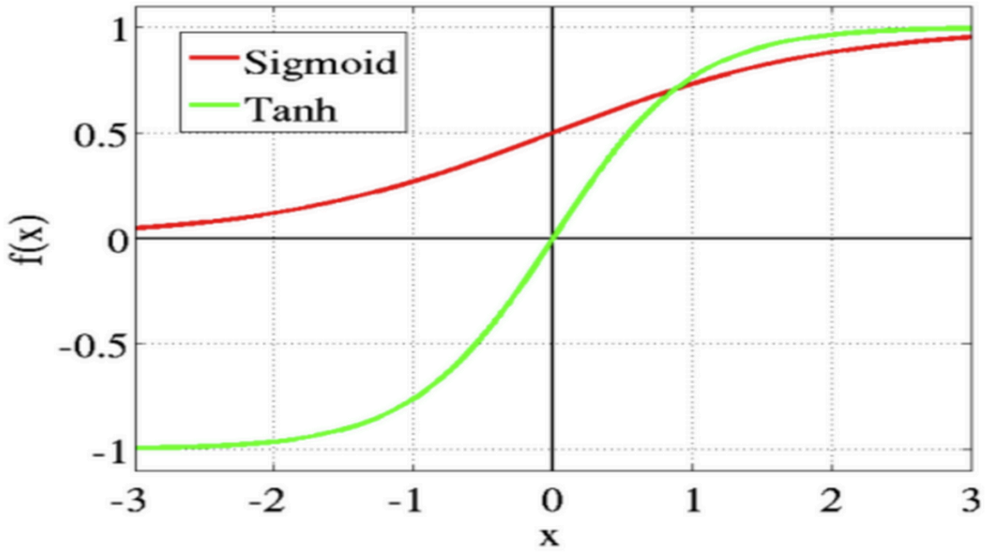



```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[-1., -2., -3., -4., -5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])


sigmoid = nn.Sigmoid()
output = sigmoid(input_tensor)
output
```

```
tensor([[[[0.2689, 0.1192, 0.0474, 0.0180, 0.0067],
          [0.9975, 0.9991, 0.9997, 0.9999, 1.0000],
          [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
          [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]]]])
```

Recap of Important Concepts and Components of CNNs

Tanh (Hyperbolic Tangent)




```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[-1., -2., -3., -4., -5.],
                               [6., 7., 8., 9., 10.],
                               [11., 12., 13., 14., 15.],
                               [16., 17., 18., 19., 20.]]]])


tanh = nn.Tanh()
output = tanh(input_tensor)
output
```

```
tensor([[[[-0.7616, -0.9640, -0.9951, -0.9993, -0.9999],
          [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000],
          [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000],
          [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000]]]])
```

Recap of Important Concepts and Components of CNNs

Softmax

```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[1.0, 2.0, 3.0, 4.0, 5.0]])

softmax = nn.Softmax()
output = softmax(input_tensor)
output
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/mo
  return self._call_impl(*args, **kwargs)
tensor([[0.0117, 0.0317, 0.0861, 0.2341, 0.6364]])
```

```python
import torch
import torch.nn as nn

input_tensor = torch.tensor([[1.0, 2.0, 3.0, 4.0, 5.0],[6.0, 7.0, 8.0, 9.0, 10.0]])

softmax = nn.Softmax(dim=0)
output = softmax(input_tensor)
output
```

```
tensor([[0.0067, 0.0067, 0.0067, 0.0067, 0.0067],
        [0.9933, 0.9933, 0.9933, 0.9933, 0.9933]])
```

```python
import torch
import torch.nn as nn

input_tensor = torch.tensor([[1.0, 2.0, 3.0, 4.0, 5.0],[6.0, 7.0, 8.0, 9.0, 10.0]])

softmax = nn.Softmax(dim=1)
output = softmax(input_tensor)
output
```

```
tensor([[0.0117, 0.0317, 0.0861, 0.2341, 0.6364],
        [0.0117, 0.0317, 0.0861, 0.2341, 0.6364]])
```

Train a SimpleCNN Model on MNIST Dataset

torch.nn.Sequential

A sequential container.

Modules will be added to it in the order they are passed in the constructor. Alternatively, an OrderedDict of modules can be passed in.

```
CLASS  torch.nn.Sequential(*args: Module) [SOURCE]
```

```
CLASS  torch.nn.Sequential(arg: OrderedDict[str, Module])
```

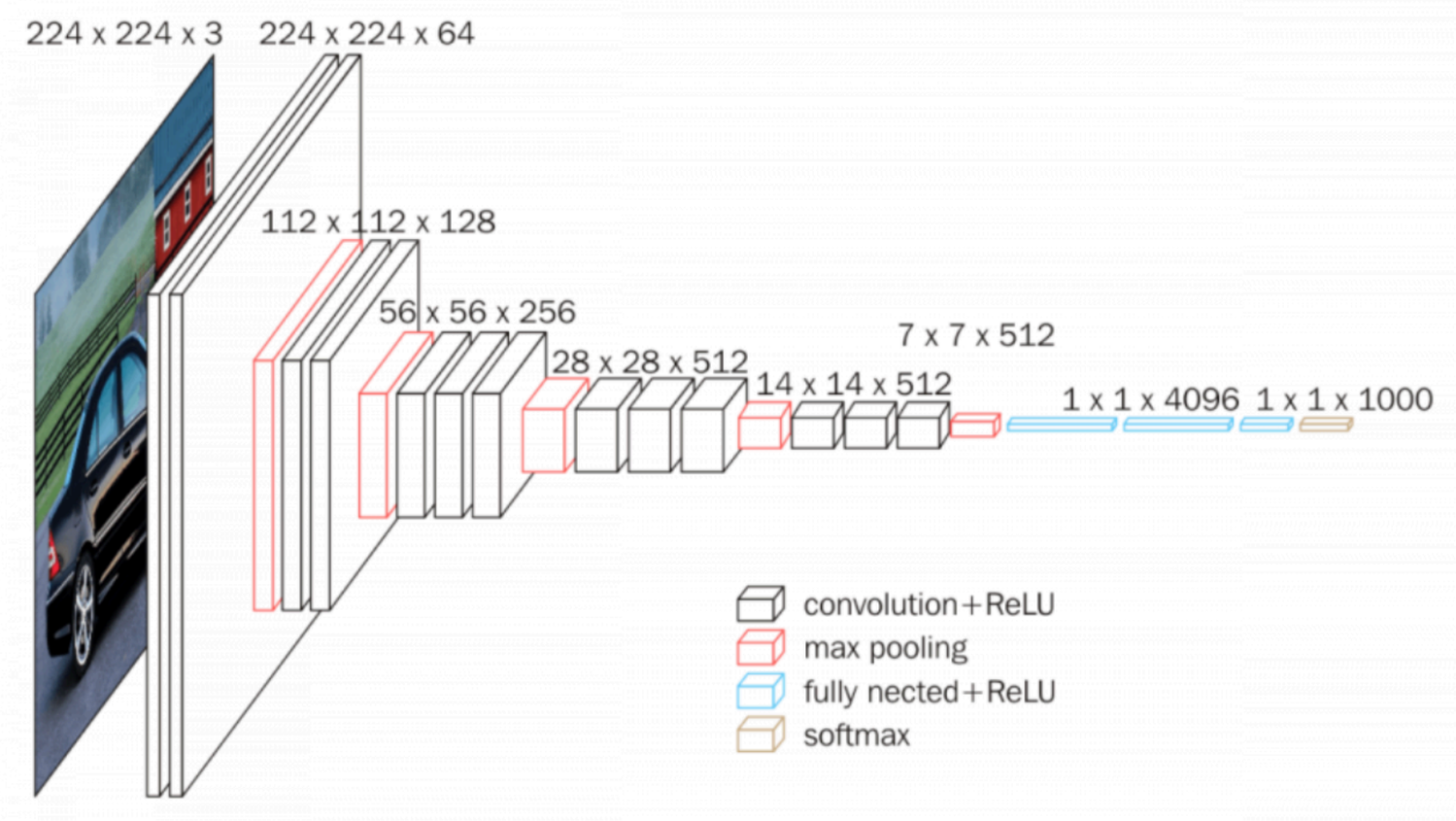
```
# Using Sequential to create a small model. When `model` is run,
# input will first be passed to `Conv2d(1,20,5)`. The output of
# `Conv2d(1,20,5)` will be used as the input to the first
# `ReLU`; the output of the first `ReLU` will become the input
# for `Conv2d(20,64,5)`. Finally, the output of
# `Conv2d(20,64,5)` will be used as input to the second `ReLU`
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Using Sequential with OrderedDict. This is functionally the
# same as the above code
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

Train a SimpleCNN Model on MNIST Dataset

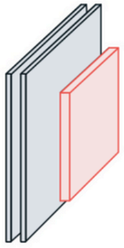
Model Description:-

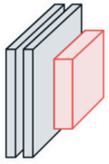
1) Convolution Layer (16 kernels (3 x 3), stride = 1 and padding = 1)
2) ReLU Activation Layer
3) Max Pooling Layer(2 x 2 kernel, stride = 2 and padding = 0)
4)Convolution Layer (32 kernels (3 x 3), stride = 1 and padding = 1)
5) Repeat steps 2-3
6) Flatten
7) Fully Connected Layer (in_channels = ?, out_channels = 128)
8) Fully Connected Layer (in_channels = 128, out_channels = 10)

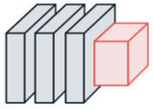
VGG16

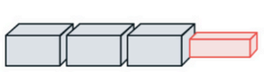

224 x 224 x 3
224 x 224 x 64
112 x 112 x 128
56 x 56 x 256
28 x 28 x 512
14 x 14 x 512
7 x 7 x 512
1 x 1 x 4096
1 x 1 x 1000
convolution+ReLU
max pooling
fully nected+ReLU
softmax

- Developed by the Visual Geometry Group (VGG) at the University of Oxford.
- Deeper than earlier networks like AlexNet. (13 conv + 3 FC Layers).
- Performed exceptionally well in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014.









CAR

Input
224 x 224 x 3

3x3 **2D Convolution**
224 x 224 x 64

3x3 **2D Convolution**
224 x 224 x 64

2D Max Pooling
112 x 112 x 64

3x3 **2D Convolution**
112 x 112 x 128

3x3 **2D Convolution**
112 x 112 x 128

2D Max Pooling
56 x 56 x 128

3x3 **2D Convolution**
56 x 56 x 256

3x3 **2D Convolution**
56 x 56 x 256

3x3 **2D Convolution**
56 x 56 x 256

2D Max Pooling
28 x 28 x 256

3x3 **2D Convolution**
28 x 28 x 512

3x3 **2D Convolution**
28 x 28 x 512

3x3 **2D Convolution**
28 x 28 x 512

2D Max Pooling
14 x 14 x 512

3x3 **2D Convolution**
14 x 14 x 512

3x3 **2D Convolution**
14 x 14 x 512

3x3 **2D Convolution**
14 x 14 x 512

2D Max Pooling
7 x 7 x 512

Flatten
25088

Dense
4096

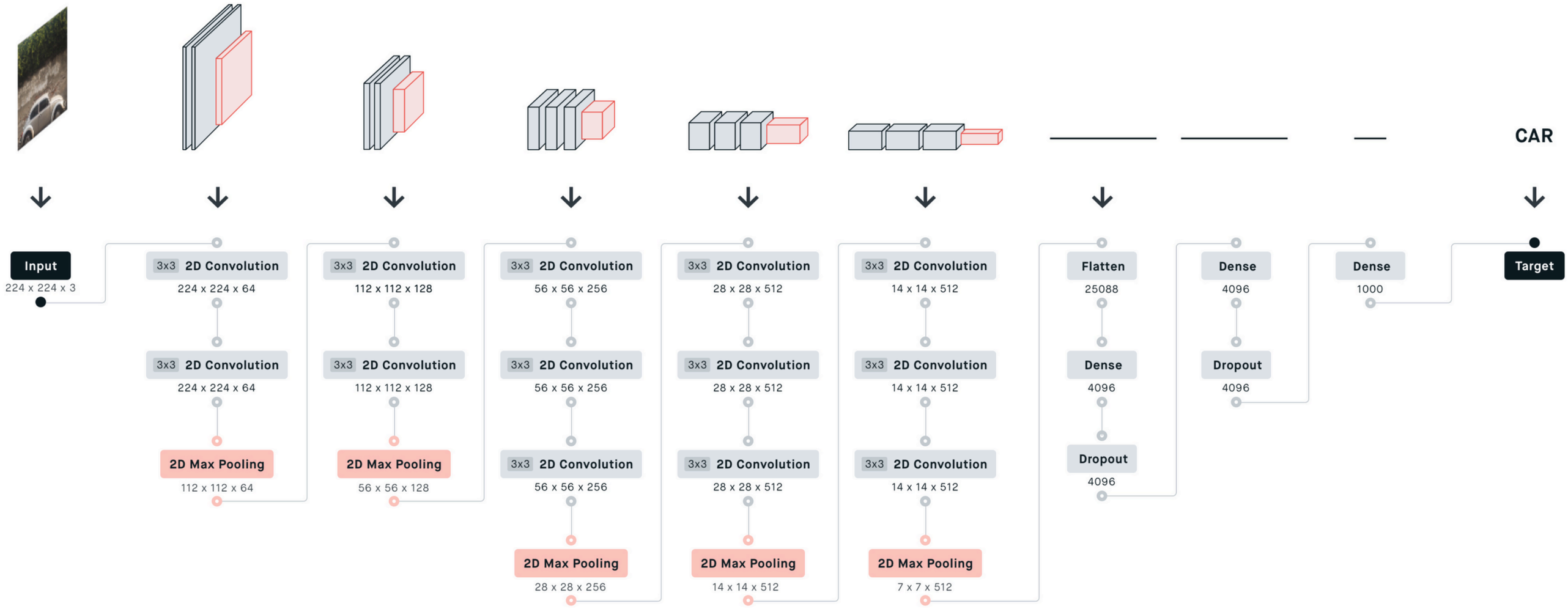
Dropout
4096

Dense
4096

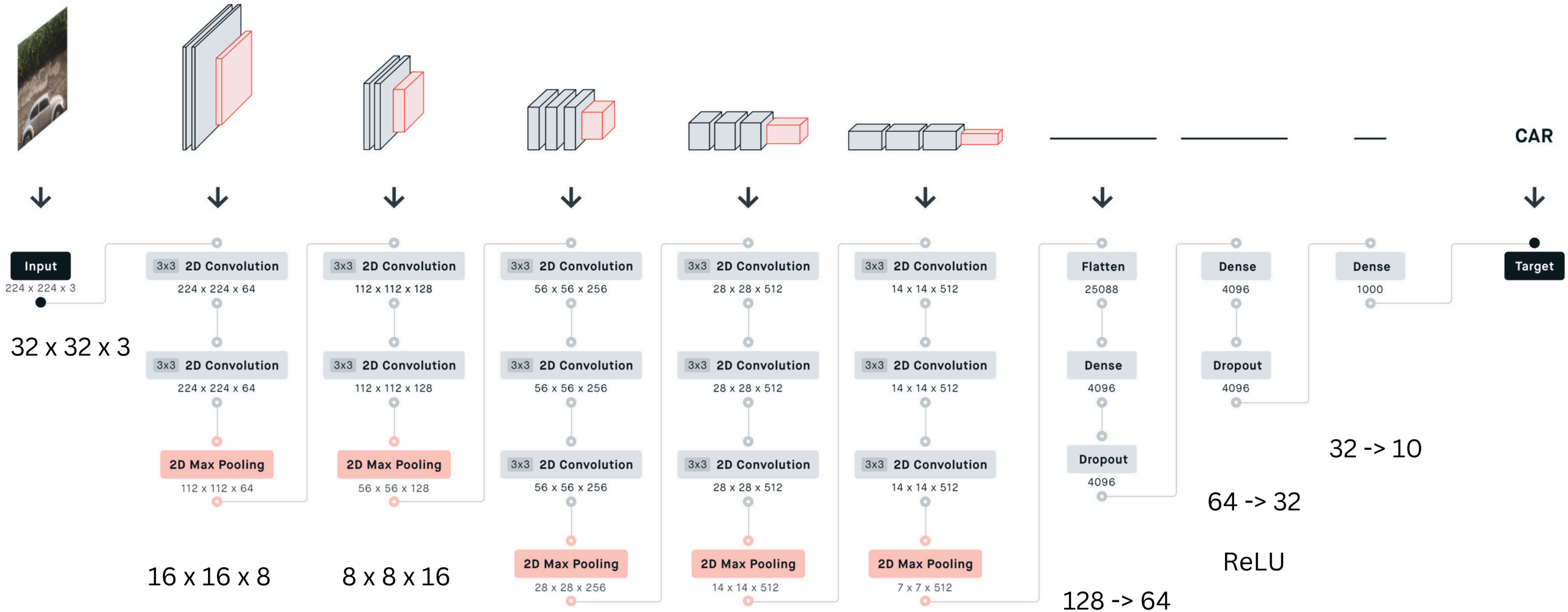
Dropout
4096

Dense
1000

Target



138M parameters


CAR
Input
224 x 224 x 3
32 x 32 x 3
3x3 2D Convolution
224 x 224 x 64
3x3 2D Convolution
224 x 224 x 64
2D Max Pooling
112 x 112 x 64
16 x 16 x 8
3x3 2D Convolution
112 x 112 x 128
3x3 2D Convolution
112 x 112 x 128
2D Max Pooling
56 x 56 x 128
8 x 8 x 16
3x3 2D Convolution
56 x 56 x 256
3x3 2D Convolution
56 x 56 x 256
3x3 2D Convolution
56 x 56 x 256
2D Max Pooling
28 x 28 x 256
4 x 4 x 32
3x3 2D Convolution
28 x 28 x 512
3x3 2D Convolution
28 x 28 x 512
3x3 2D Convolution
28 x 28 x 512
2D Max Pooling
14 x 14 x 512
2 x 2 x 64
3x3 2D Convolution
14 x 14 x 512
3x3 2D Convolution
14 x 14 x 512
3x3 2D Convolution
14 x 14 x 512
2D Max Pooling
7 x 7 x 512
1 x 1 x 128
Flatten
25088
Dense
4096
Dropout
4096
128 -> 64
ReLU
Dropout
Dense
4096
Dropout
4096
64 -> 32
ReLU
Dense
1000
32 -> 10
Target

CIFAR10 Dataset

Classification Dataset.

Consists of 60,000 color images (50,000 train + 10,000 test)

10 Classes (6,000 images per class)

torch.nn.Dropout

During training, randomly zeroes some of the elements of the input tensor with probability p.

p is set to 0.5 by default.

CLASS torch.nn.Dropout(*p=0.5, inplace=False*) [SOURCE]

U-Net

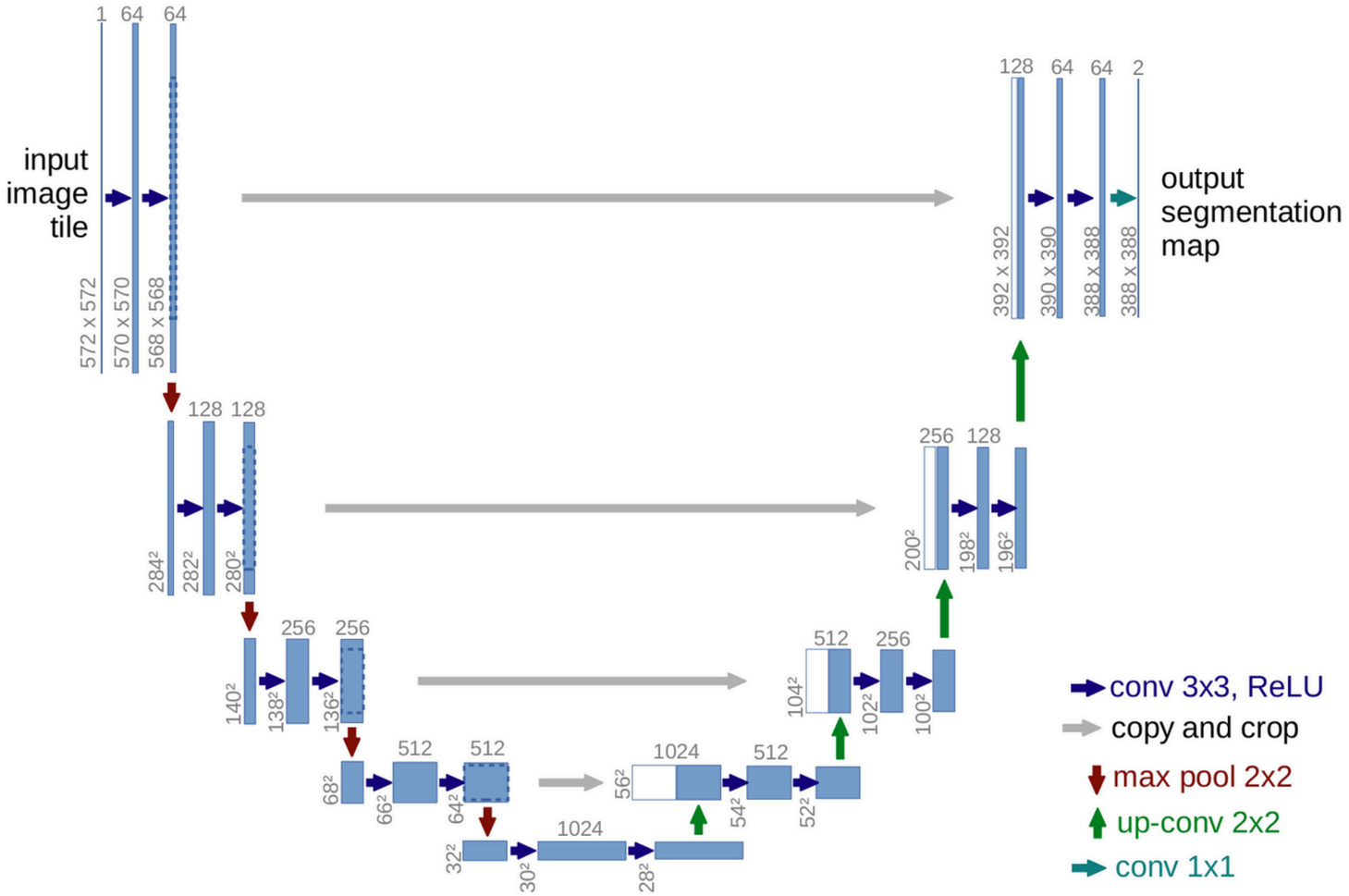


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Network Architecture

- The architecture consists of a contracting path (left side) and an expansive path (right side).
- The contracting path follows the typical architecture of a convolutional network.
- It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling.
- At each downsampling step we double the number of feature channels.

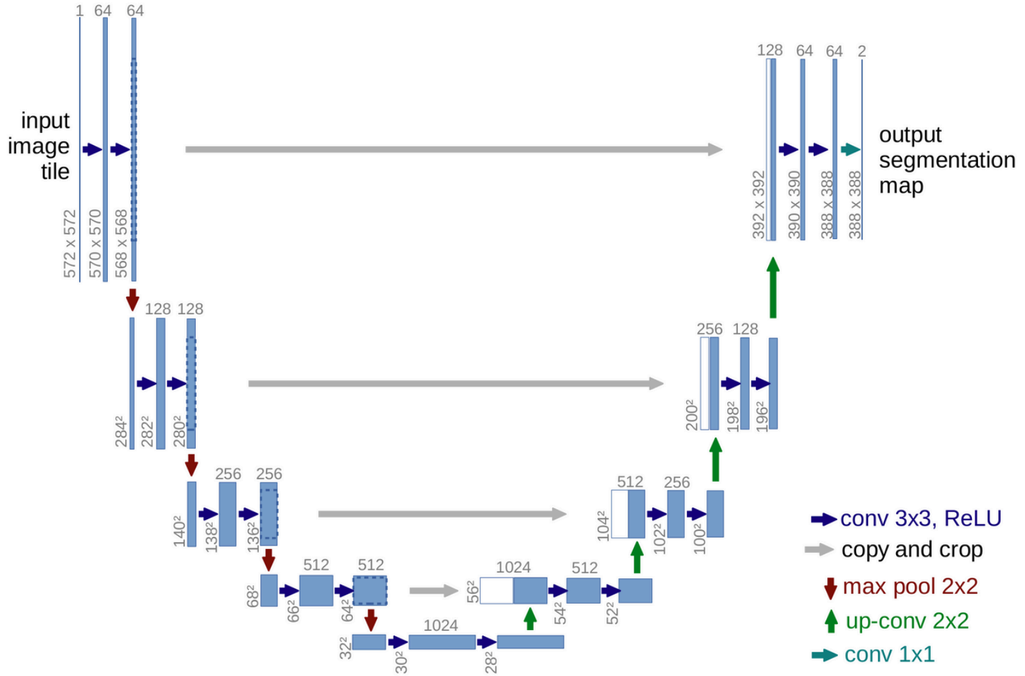


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

- Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (up-convolution) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU.
- The cropping is necessary due to the loss of border pixels in every convolution.
- At the final layer a 1x1 convolution is used to map each 64 component feature vector to the desired number of classes.
- In total the network has 23 convolutional layers.

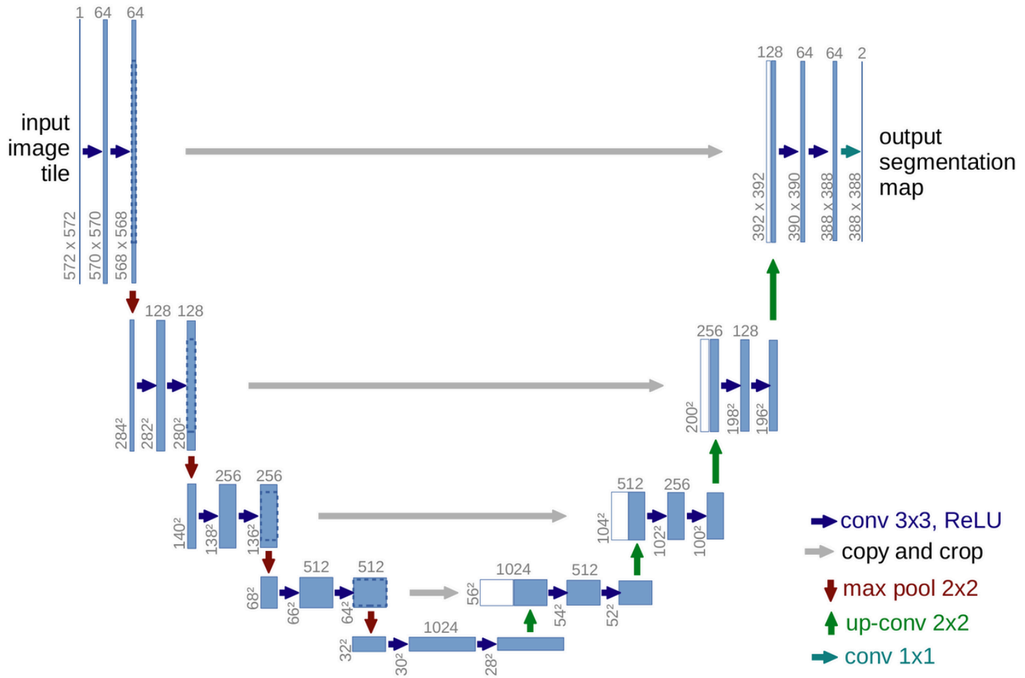


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Train a U-Net Model on Brain Tumour (Semantic Segmentation) Data

The architecture presented in the paper consists of unpadded convolutions.

We, however, for our own convenience will use padded convolutions.

Creating U-Net Model Class

Hints :-

Note the repeated pattern of double convolutions at every step.
Makes sense to define a function for that.

torch.cat function is used to concatenate two tensors (described on the next slide)

torch.cat

Concatenates the given sequence of seq tensors in the given dimension.

All tensors must either have the same shape (except in the concatenating dimension).

```
torch.cat(tensors, dim=0, *, out=None) → Tensor
```

ResNet

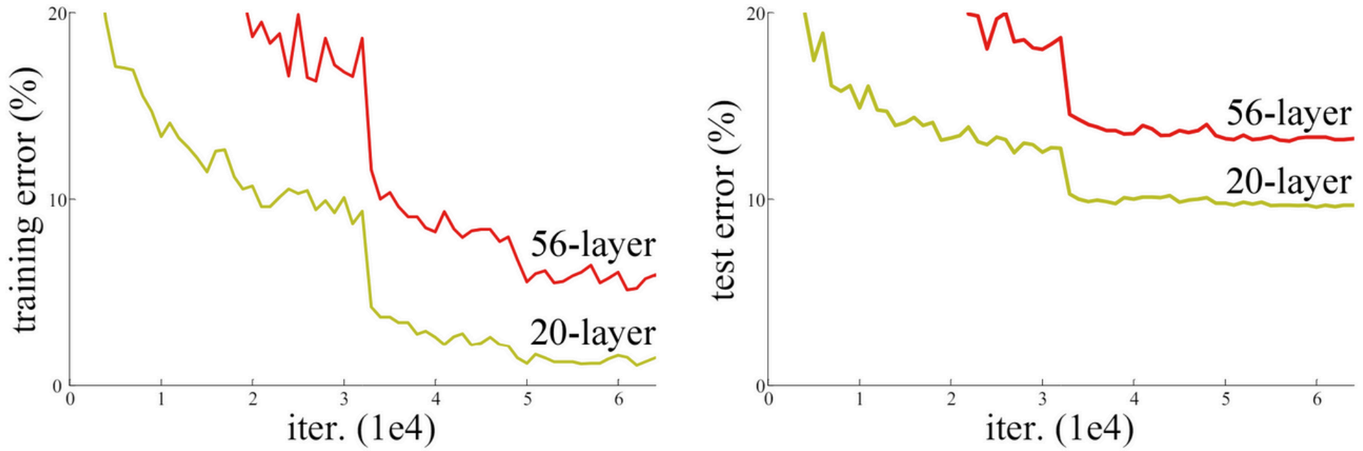


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

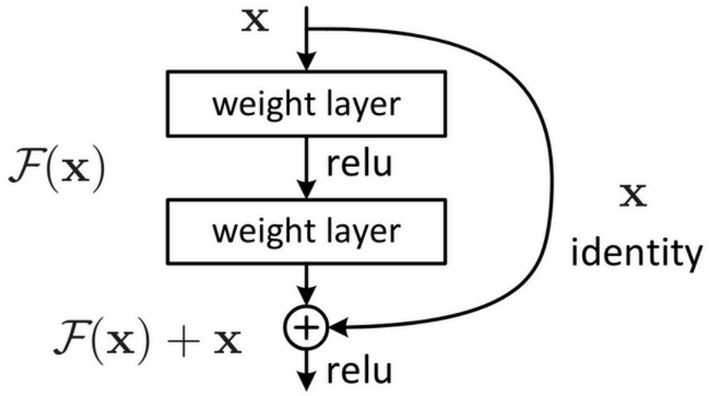
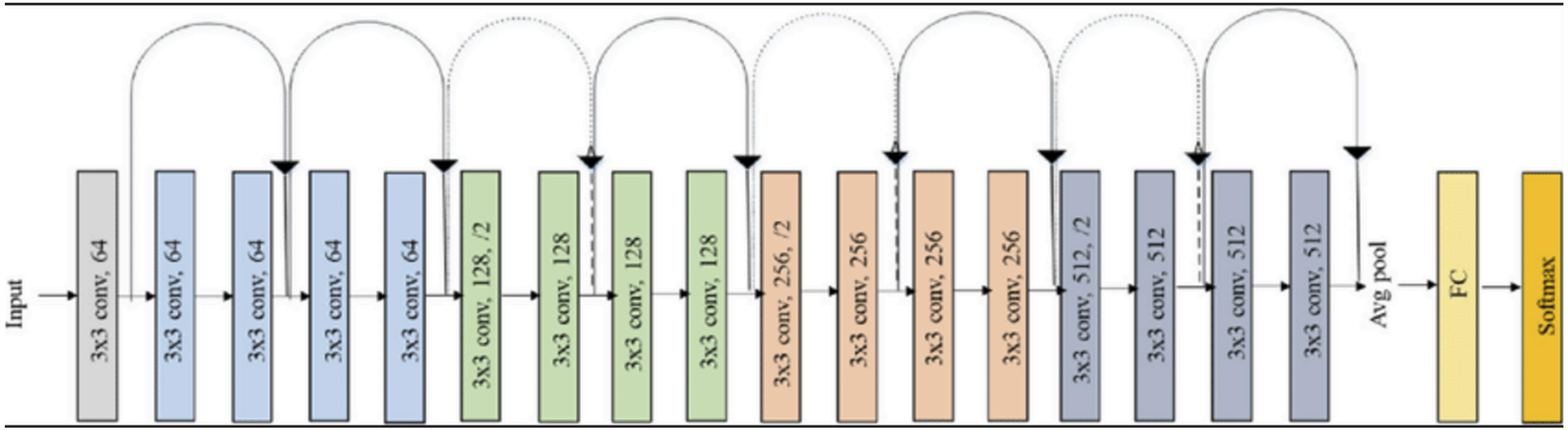


Figure 2. Residual learning: a building block.



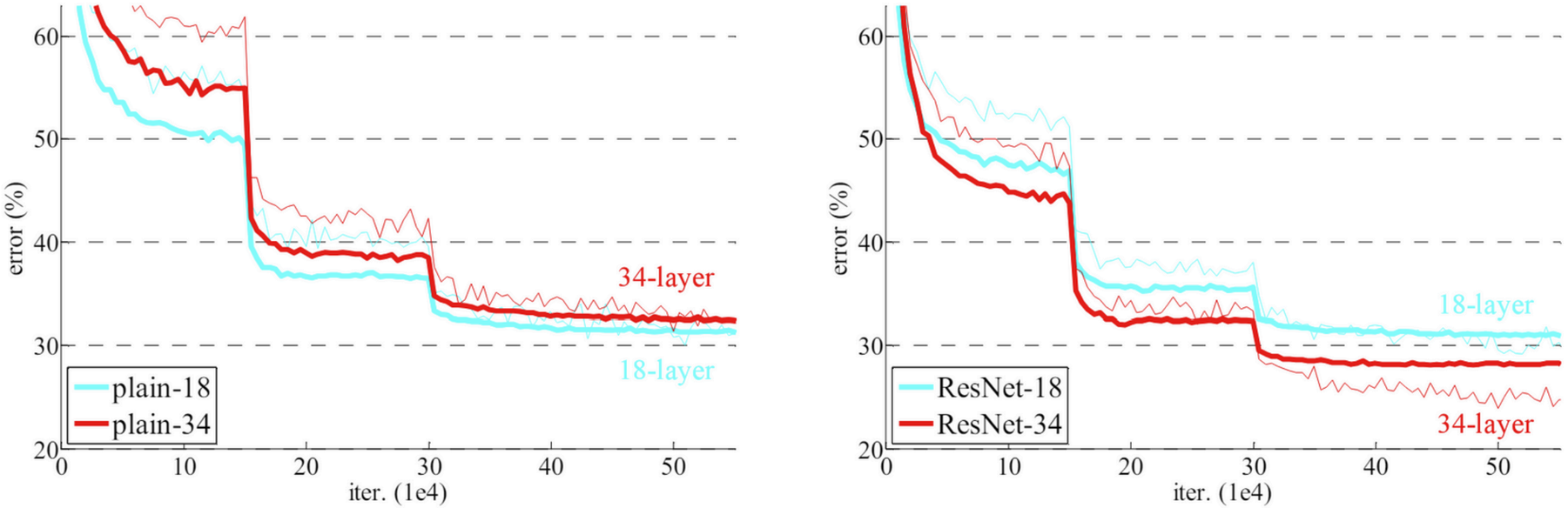


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Thank You